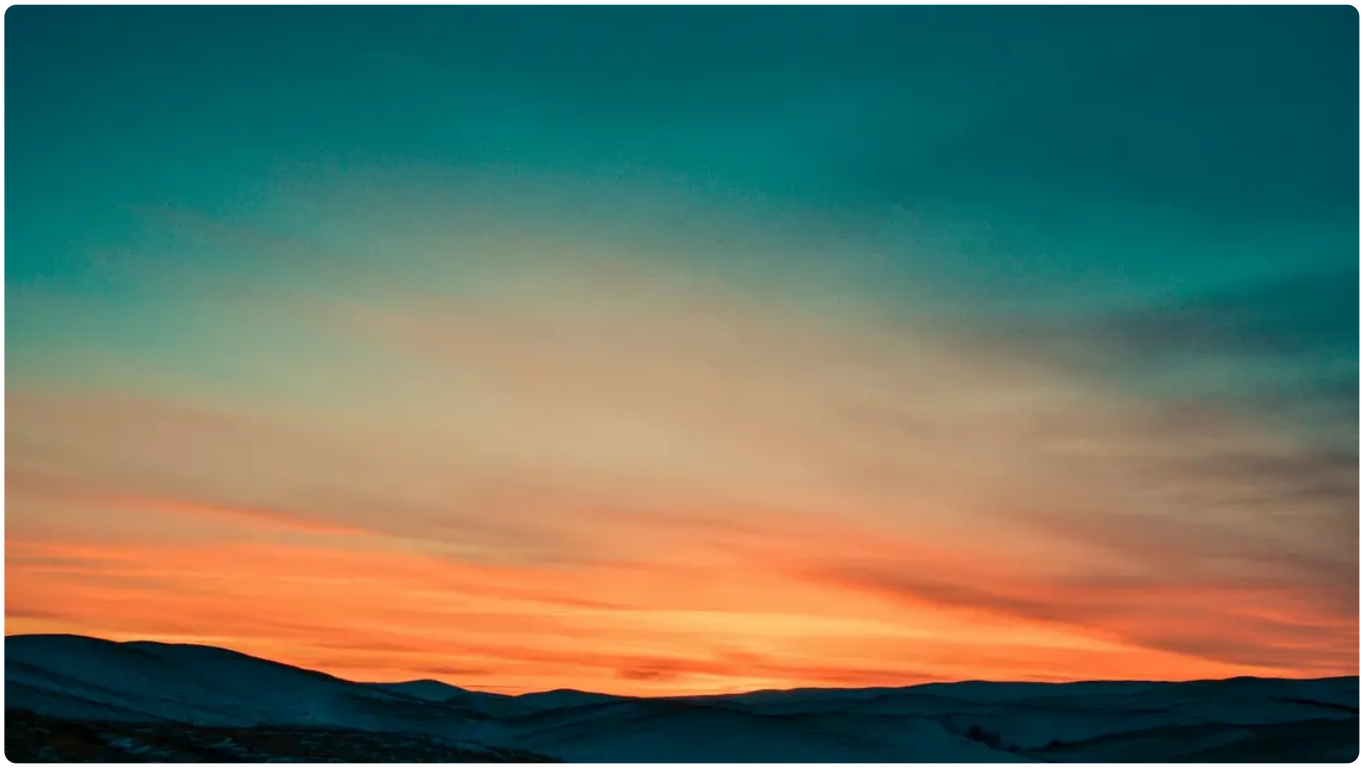
下载APP

09 | 压测模型：如何建立一套完整的全链路压测模型？

2021-11-06 高楼

《全链路压测实战30讲》 课程介绍 >



讲述：高楼

时长 21:38 大小 19.81M

你好，我是高楼。

这一讲，我们开始学习模型部分。

在完整的性能工程中，模型到底有哪些内容呢？这个问题，估计每一个做性能项目的人都没有看到过什么定论。

领资料

这节课呢，我想跟你分享一下我总结出来的性能项目中的模型，我把它称为**性能五模型**

简单解释一下，它分为**业务模型、容量模型、监控模型、失效模型**和**分析模型**五个部分。

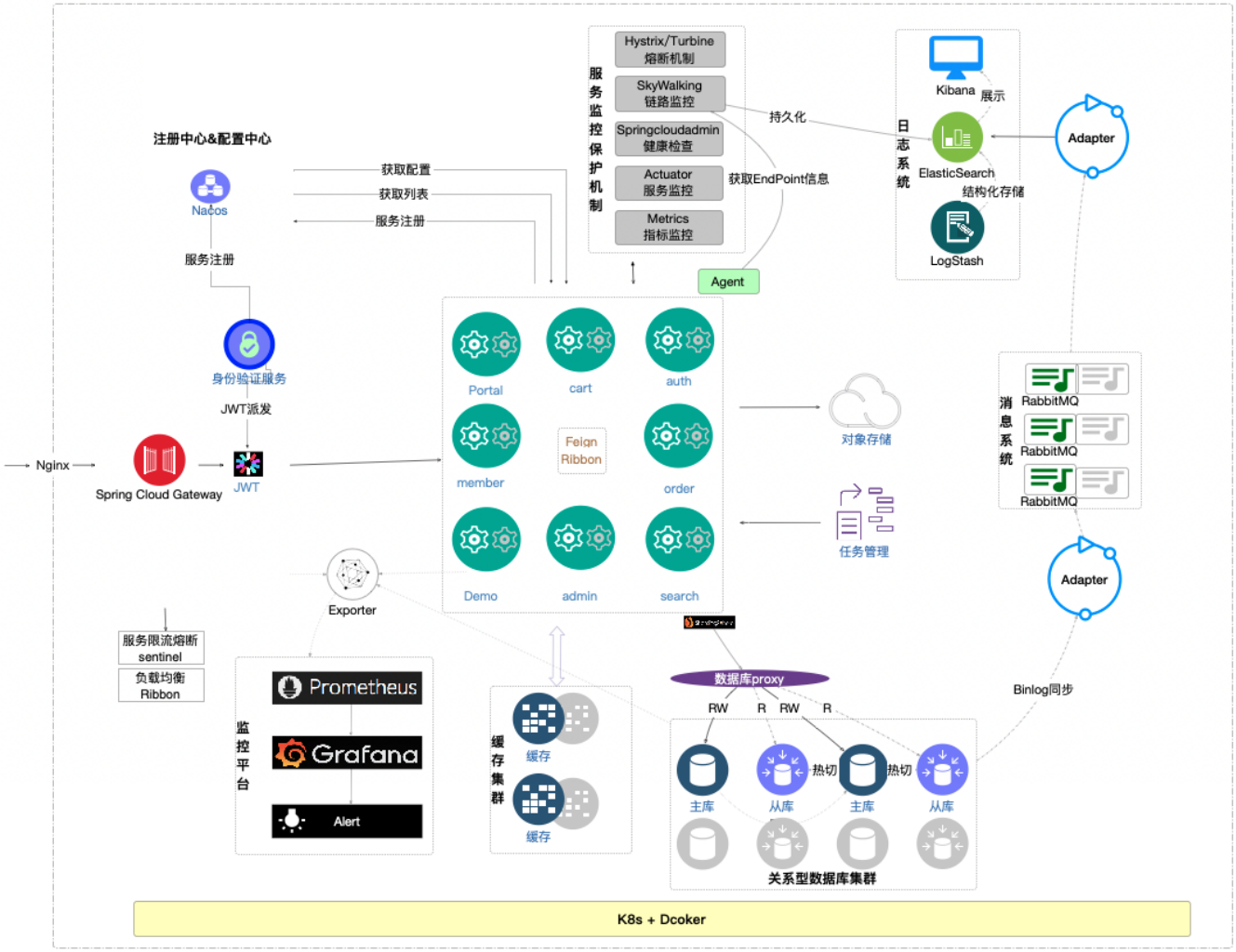
1. 业务模型确定了压力工具中的设置；
2. 容量模型确定了系统的设计容量计算方式；
3. 监控模型确定了监控的内容；
4. 分析模型确定了分析的逻辑；
5. 失效模型确定了异常场景。

那么，在全链路压测项目中，这些模型产生了什么变化呢？

从分类和内容上来说，没有任何变化。主要的变化是，所有的模型都乘以了 2。

因为全链路压测强调的是**线上执行**，所以不管你用什么样的方式来组织线上的架构，除了正在运行的真实模型之外，你还要从全链路压测的视角把这些模型再考虑一遍。

在具体描述每个模型之前，我先把整体的架构图展示一下，以便让你看到我们是在哪些地方做了改造的动作。



在这个架构图中，我们对微服务、缓存、队列、数据库做了改造，用灰色标示的是做了改造的部分。其中，微服务、缓存、队列部分没有用新的硬件设备，是在同样的硬件中实现的；数据库创建了影子库，所以增加了同配置的硬件。

现在我们来说一下每一个模型的具体内容。

业务模型

业务模型就是将真实场景的业务比例复制或推算出来的过程。关于复制，我们要考虑不同的手段。

手段一：线上流量复制的方式复制出业务模型。

简而言之就是流量复制，这也是现在全链路压测中经常讨论的一个话题。流量复制使用的工具，我在第 6 讲中已经有过详细的论述，如果你记不清了，也可以回过头去温习一下。这里就不复述了。

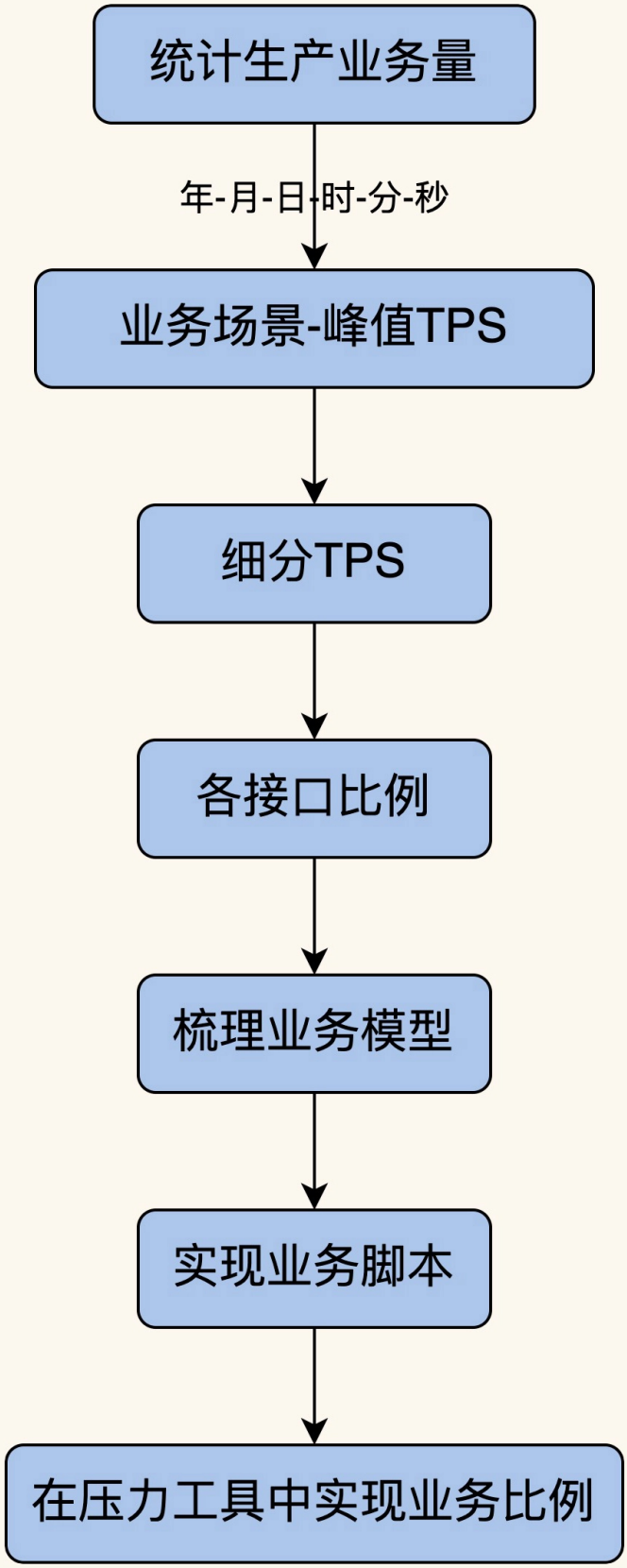
这里我们要讨论的是：复制出来的流量，是不是回放或放大回放就可以满足我们的目标了呢？

当然没有那么简单，因为我们先要确定**目标**是什么。如果我们的目标就是要测试录制出来的那一段流量产生的业务比例，这样做确实是可以的。

但如果我们的目标是通过放大录制的流量来评估未来的场景，那基本上就靠不住了。因为复制出来的业务模型，是不能代表未来业务模型的变化的。

手段二：通过统计线上流量的方式复制出业务模型。

我把详细的流程给你梳理了一下：



只要业务比例和生产环境一致，通过这种方式复制出的业务模型，就和录制回放手段产生的压力结果没有区别。

不过，如果要想推算未来的模型，也存在同样的弊端。

谈到这里，我们就不得不说说推算业务模型这件事了。假设我们拿到了这样的生产业务模型。

序号	操作	接口	业务比例
1	打开首页	/home/content	24%
2	用户登陆	/sso/login (这个接口还会调用/oauth/token、/sso/loadByUsername)	11%
3	用户信息查询	/member/address/list	11%
4	商品加入购物车	/cart/add	11%
5	查询购物车信息	/cart/list	7%
6	购物车信息确定订单	/order/generateConfirmOrder	7%
7	生成订单信息	/order/generateOrder	7%
8	支付前查询订单列表	/order/list	7%
9	支付订单信息	/order/paySuccess	7%
10	支付后查询订单详情	/order/detail/${orderId}	11%

极客时间

但是由于未来业务的变化，业务部门评估认为，未来购买的比例会增加。那从接口上来说，就是图表中 4-9 之间的业务比例会增加，于是我们得到了下面这个模型。

序号	操作	接口	业务比例
1	打开首页	/home/content	17%
2	用户登陆	/sso/login (这个接口还会调用/oauth/token、/sso/loadByUsername)	10%
3	用户信息查询	/member/address/list	10%
4	商品加入购物车	/cart/add	10%
5	查询购物车信息	/cart/list	8%
6	购物车信息确定订单	/order/generateConfirmOrder	8%
7	生成订单信息	/order/generateOrder	8%
8	支付前查询订单列表	/order/list	8%
9	支付订单信息	/order/paySuccess	8%
10	支付后查询订单详情	/order/detail/${orderId}	10%

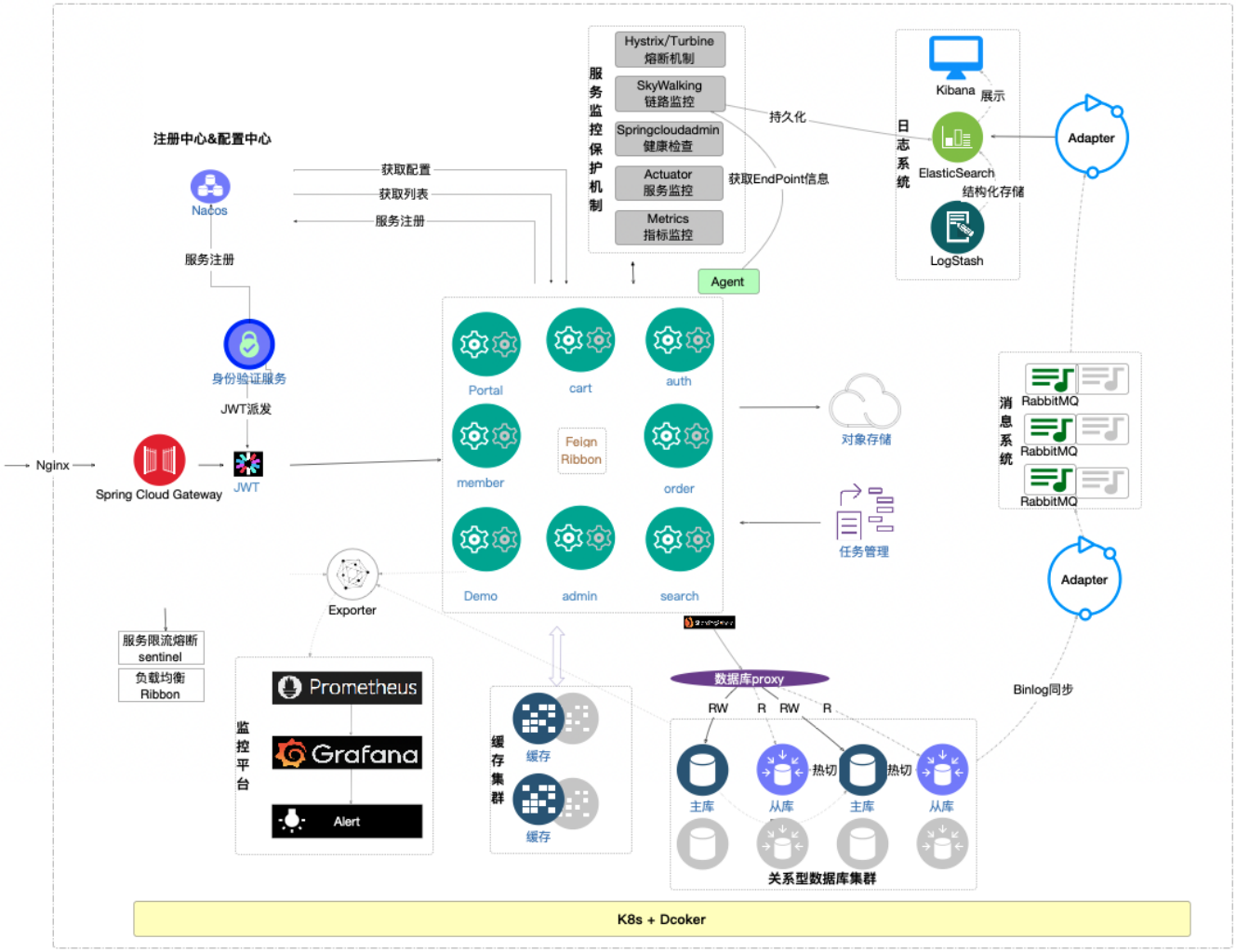
极客时间

如果我们想实现这样的业务模型，那用复制出来的流量回放显然是不可以的。要实现这样的业务模型大概只有一个选择，那就是用压力工具来实现压力。

所以，**对于业务模型来说，关键是要看测试目标。**

容量模型

在执行测试场景之前，要先计算一下系统大概能支持多少容量（本节课后面的容量都用 TPS 这个概念来承载）。容量模型的计算离不开架构图，这张图我在前面已经给你展示过了，我再放在这里加深一下你的印象，希望你能把它牢牢地记下来。



有了架构图，我们再来输入一些数据。

1. 微服务应用在总资源 40C80G 的硬件资源，可以支撑混合场景 1500TPS，且硬件资源已饱和。

2. 在上述条件下，2 台数据库 (8C16G) 使用率最大的硬件资源 CPU，只用到了 50% 左右；2 台缓存（4C8G）和 2 台队列服务器（4C8G）最大的资源使用率（CPU 使用率）只有 20% 左右；2 台网关（4C8G）最大资源使用率是 40%。

请注意，上面的信息并不需要臆测，只要通过正常的性能容量场景就可以得到这样的数据。那么，如果现在我们希望系统支持 3000TPS，需要多少资源呢？

如果用线性计算，我们可以得到图片中这些变化。

类型	1500TPS的硬件资源	3000TPS的硬件资源
网关	4C8G x 2	4C8G x 2
微服务应用	40C80G	80C160G
数据库服务器	8C16G x 2	8C16G x 2
缓存服务器	4C8G x2	4C8G x2
队列服务器	4C8G x2	4C8G x2

极客时间

也就是说，应用类的资源要增加一倍，而数据库、缓存、队列服务器的资源可以不用增加。

可是，真的就这么简单吗？你是不是也觉得这样的计算非常不靠谱？

试想一下，当压力增加的时候，数据库资源用完，响应会变慢，那显然 TPS 就会下降，所以不可能达到 3000TPS。而缓存和队列服务器就还好，因为还有比较大的空间。

如果这么做不合理，那我们是不是再增加一些数据库服务器资源就好了呢？其实即便是这样，我们也无法让系统容量成倍地增加，因为还有另一个问题：微服务应用资源已经用完了。

如果我们增加了成倍的资源，在不考虑其他服务器的情况下，这成倍资源的增加会带来 TPS 的成倍增加吗？这是不能确定的。因为资源的增加，会带来服务节点的增加，紧接着

会带来远程调用时网络带宽的增加。请注意，这个增加不止是网络资源，还包括网关和负载均衡压力的增加。

所以我不建议这样线性地计算，怎么来计算呢？最好的方式就是拿真实的数据说话，采用递增加压的方式，先看一下 TPS 增加的幅度和资源消耗之间的关系。相应地创建下面这样的表格：

类型	1500TPS的硬件资源	2000TPS的硬件资源	2500TPS的硬件资源	3000TPS的硬件资源
网关	4C8G x 2	?	?	?
微服务应用	40C80G	?	?	?
数据库服务器	8C16G x 2	?	?	?
缓存服务器	4C8G x2	?	?	?
队列服务器	4C8G x2	?	?	?

极客时间

我们先增加到 2000TPS，再增加到 2500TPS，再增加到 3000TPS，然后看看硬件资源的增长率。这里我只把硬件的增加列出来，就不再详细记录使用率的百分比了。

类型	1500TPS的硬件资源	2000TPS的硬件资源	2500TPS的硬件资源	3000TPS的硬件资源
网关	4C8G x 2	4C8G x 2	5C8G x 2	6C8G x 2
微服务应用	40C80G	50C100G	70C140G	100C200G
数据库服务器	8C16G x 2	9C16G x 2	11C20G x 2	14C16G x 2
缓存服务器	4C8G x2	4C8G x2	4C8G x2	4C8G x2
队列服务器	4C8G x2	4C8G x2	4C8G x2	4C8G x2

极客时间

通过数据来看，网关的资源增加主要是 CPU，微服务应用和数据库也都有增加，但并不是完全线性的，TPS 越往上增加，微服务应用和数据库的增幅也越大，成本越高。缓存和队列由于没有达到使用的上限，所以不需要增加。

在这样的示例中，我们得到最重要的启发就是：**容量的规划不能通过简单的线性计算得到合理的结果，而是要通过容量场景的执行得到一个合理的增长模型。**

监控模型

监控模型在全链路压测过程中是非常重要的模型。根据 RESAR 性能工程理念，全局监控的关键是“全”。怎么才能全呢。那就要在监控之前做几个动作：

1. 分析技术栈。我这里主要列出了涉及到改造的技术栈。

类型	技术栈
网关	Spring Cloud Gateway
微服务应用	SpringBoot
缓存	Redis
队列	RabbitMQ
数据库	MySQL

极客时间

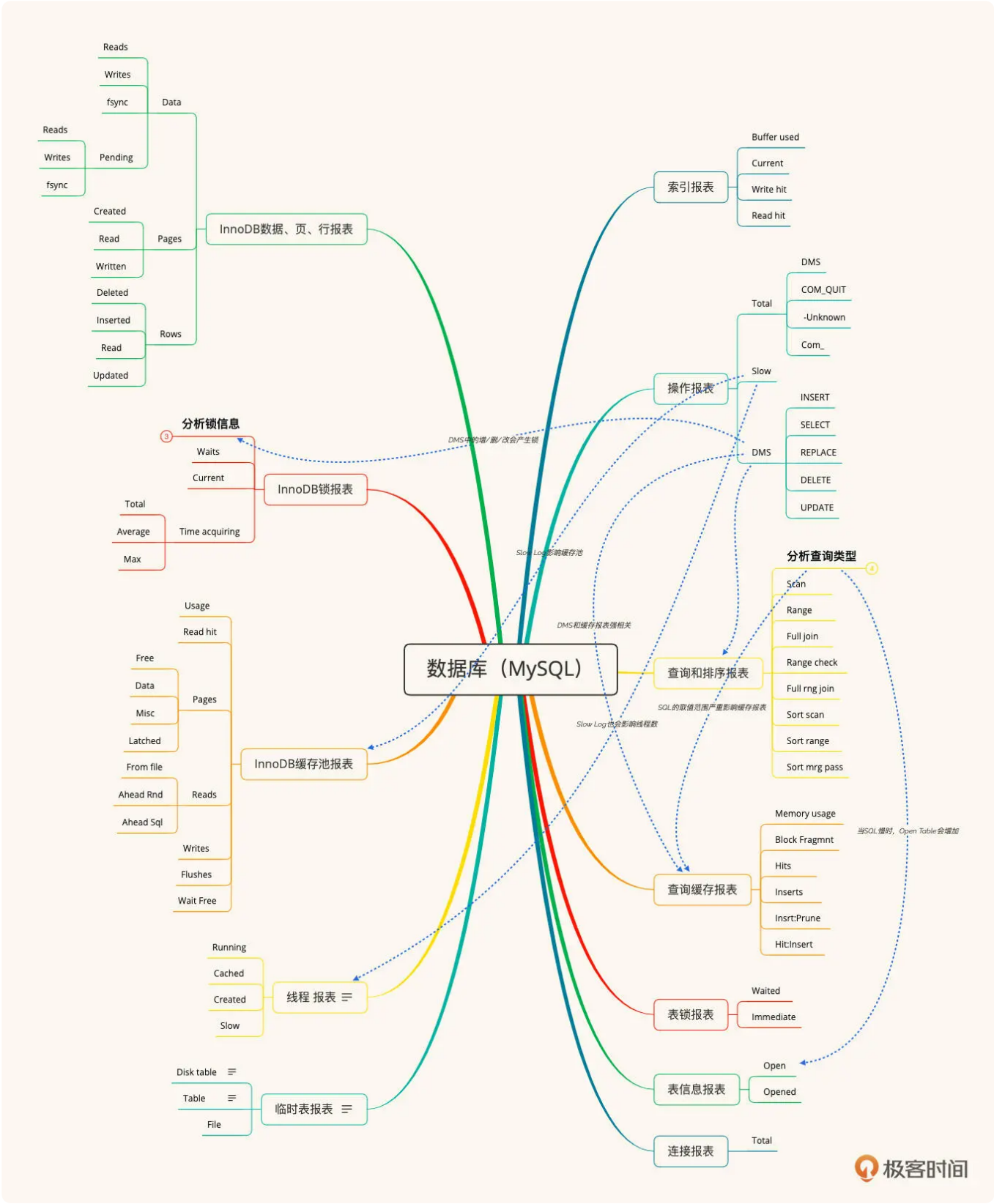
2. 创建性能分析决策树。

接下来，我们要为上面提到的网关、微服务应用、缓存、队列和数据库一一创建性能分析决策树。

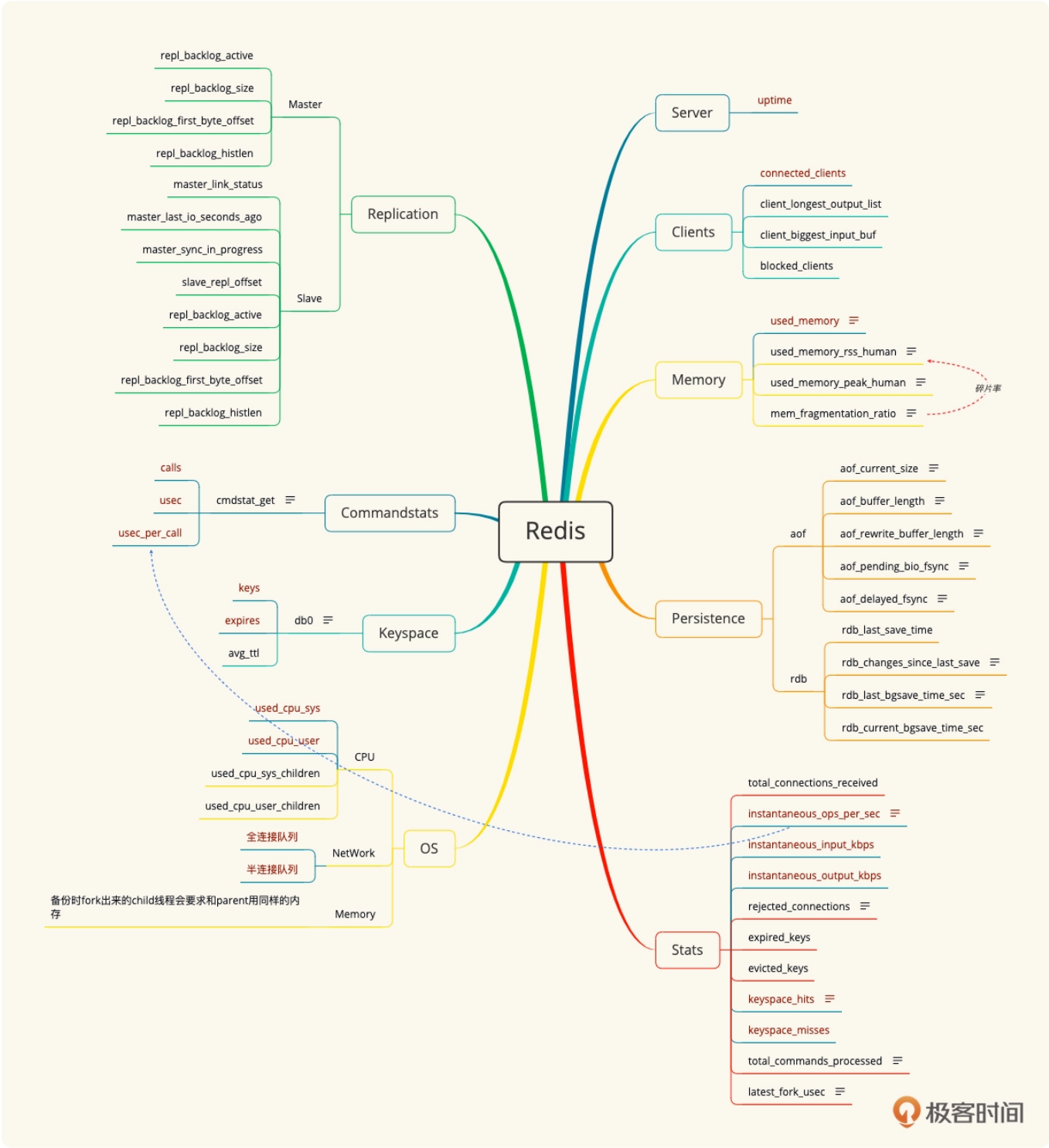
由于网关和微服务应用都是基于 Java 编写的，所以这里我们直接创建一个 Java 应用的性能分析决策树：

Java
Classes
Total loaded
Total unloaded
Current Classes loaded
GC
gclog
Total Collection Time
Collection Count
GC Duration
GC Thread Count
Garbage collector
G1
CMS
堆的使用和GC效率关系密切
GC消耗掉的CPU
Memory
Current
Min
Max
Heap Usage
Heap
PS Eden Space
PS Old Gen
Ps Survivor Space
Committed Java Heap
Code Cache
Compressed Class Space
Metaspace
Java Thread Stack
Native Stack
NonHeap Usage
CPU
CPU usage
GC activity
CPU Machine
Running状态的线程消耗CPU
OS
CPU
Memory
Disk
Network
System
线程调度会引发CS和softirq变高
Thread
CurrentThread
Running
Sleeping
Wait
Park
Monitor
CurrentThreadCpuTime
CurrentThreadCpuTimeSupported
CurrentThreadUserTime
DaemonThreadCount
DeadlockedThreadCount
MonitoredDeadlockedThreadCount
ObjectMonitorUsageSupported
PeakThreadCount
ThreadCount
ThreadCpuTimeEnabled
ThreadCpuTimeSupported
TotalStartedThreadCount
极客时间

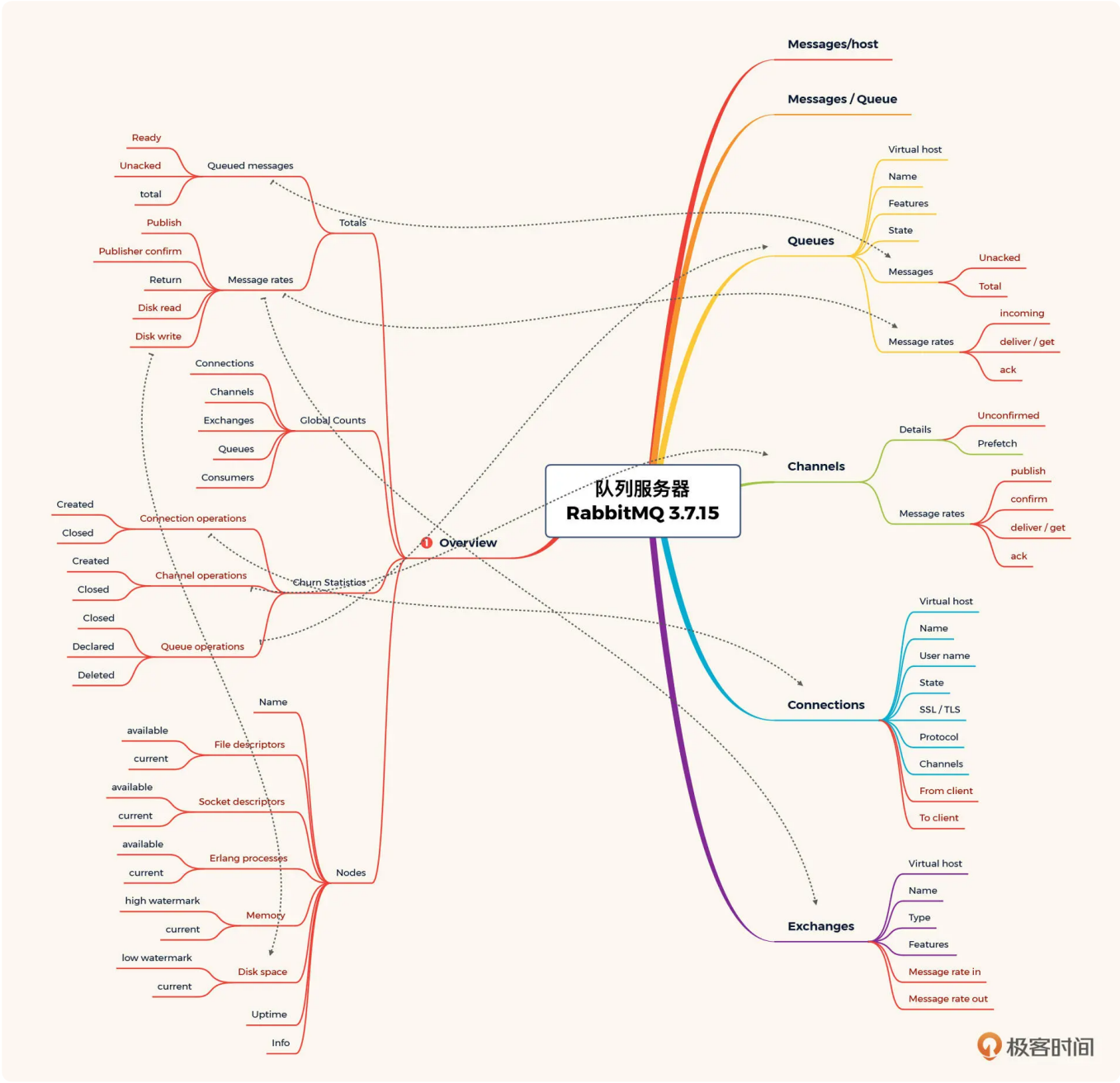
数据库性能分析决策树：


数据库（MySQL）
InnoDB数据、页、行报表
Data
Reads
Writes
fsync
Pending
Reads
Writes
fsync
Pages
Created
Read
Written
Rows
Deleted
Inserted
Read
Updated
InnoDB锁报表
分析锁信息
Waits
Current
Time acquiring
Total
Average
Max
InnoDB缓存池报表
Usage
Read hit
Pages
Free
Data
Misc
Latched
Reads
From file
Ahead Rnd
Ahead Sql
Writes
Flushes
Wait Free
线程 报表
Running
Cached
Created
Slow
临时表报表
Disk table
Table
File
索引报表
Buffer used
Current
Write hit
Read hit
操作报表
Total
DMS
COM_QUIT
-Unknown
Com_
Slow
DMS
INSERT
SELECT
REPLACE
DELETE
UPDATE
查询和排序报表
分析查询类型
Scan
Range
Full join
Range check
Full rng join
Sort scan
Sort range
Sort mrg pass
查询缓存报表
Memory usage
Block Fragmnt
Hits
Inserts
Insrt:Prune
Hit:Insert
表锁报表
Waited
Immediate
表信息报表
Open
Opened
连接报表
Total
DMS中的增/删/改会产生锁
Slow Log影响缓存池
DMS和缓存报表强相关
SQL的取值范围严重影响缓存报表
Slow Log也会影响线程数
当SQL慢时，Open Table会增加
极客时间

缓存性能分析决策树：

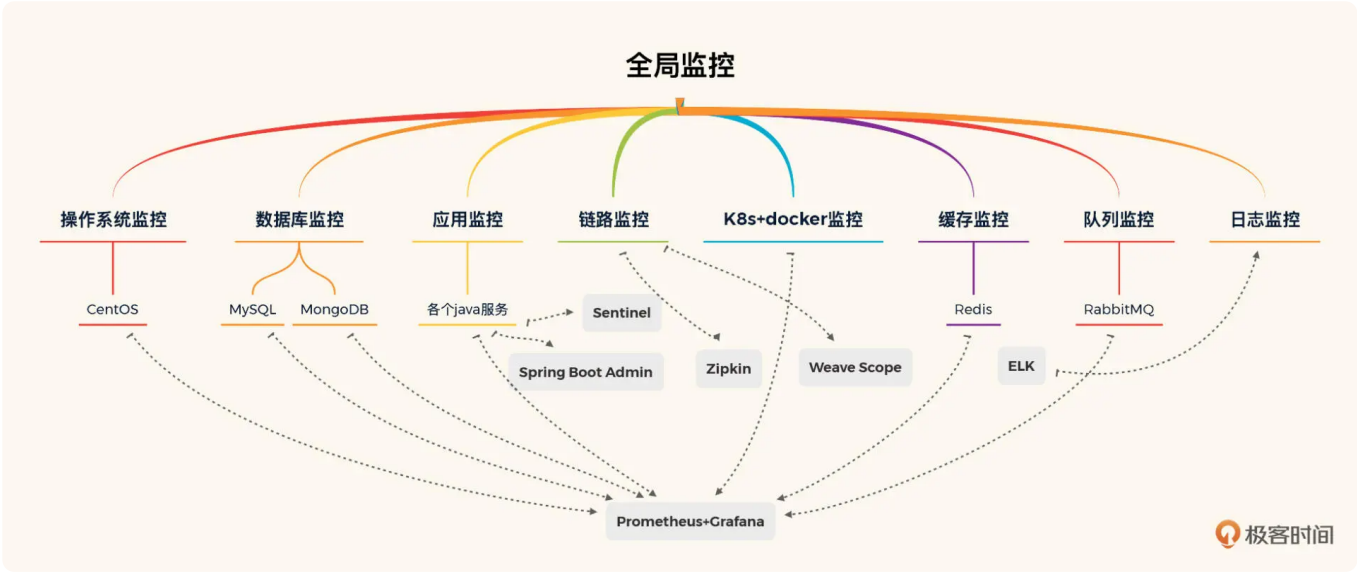

Redis
Replication
Master
repl_backlog_active
repl_backlog_size
repl_backlog_first_byte_offset
repl_backlog_histlen
Slave
master_link_status
master_last_io_seconds_ago
master_sync_in_progress
slave_repl_offset
repl_backlog_active
repl_backlog_size
repl_backlog_first_byte_offset
repl_backlog_histlen
Server
uptime
Clients
connected_clients
client_longest_output_list
client_biggest_input_buf
blocked_clients
Memory
used_memory
used_memory_rss_human
used_memory_peak_human
mem_fragmentation_ratio
碎片率
Persistence
aof
aof_current_size
aof_buffer_length
aof_rewrite_buffer_length
aof_pending_bio_fsync
aof_delayed_fsync
rdb
rdb_last_save_time
rdb_changes_since_last_save
rdb_last_bgsave_time_sec
rdb_current_bgsave_time_sec
Stats
total_connections_received
instantaneous_ops_per_sec
instantaneous_input_kbps
instantaneous_output_kbps
rejected_connections
expired_keys
evicted_keys
keyspace_hits
keyspace_misses
total_commands_processed
latest_fork_usec
Commandstats
cmdstat_get
calls
usec
usec_per_call
Keyspace
db0
keys
expires
avg_ttl
OS
CPU
used_cpu_sys
used_cpu_user
used_cpu_sys_children
used_cpu_user_children
NetWork
全连接队列
半连接队列
Memory
备份时fork出来的child线程会要求和parent用同样的内存
极客时间

队列性能分析决策树：



这样，我们几个关键技术组件的性能分析决策树就完成了。其他的技术栈，你可以自己创建一下，组织的形式和分类的方式并不是固定的，只要你能理解就可以。关键在于监控数据的完整性。

3. 有了整个项目的性能分析决策树之后，就要把这些需要监控的计数器，都体现到监控工具中了。对应每一个技术组件，我们用下图表示一下覆盖关系。



看完这张图你会发现，我们在设计监控模型的时候，是每一个技术组件都会覆盖到的。

这个逻辑在所有的性能项目中都是一样的。

而在全链路压测的项目中，由于做了改造，必然会涉及到一些监控计数器的增加。具体有哪些呢？我们还是拿关键组件来说明。

类型	技术栈	是否增加监控	理由
网关	Spring Cloud Gateway	不需要	没有增加新的组件节点，改造是在同一进程中完成的。
微服务应用	SpringBoot	不需要	没有增加新的组件节点，改造是在同一进程中完成的。
缓存	Redis	不需要	没有增加新的组件节点，改造是通过增加有标识的key完成的。
队列	RabbitMQ	不需要	没有增加新的组件节点，改造是通过增加有标识的消息完成的。
数据库	MySQL	需要	使用了新硬件和影子库，增加了组件节点。

极客时间

从表格上可以看出，在我们这个全链路压测项目中，相比较之前的监控内容，只是增加了新的数据库节点而已。

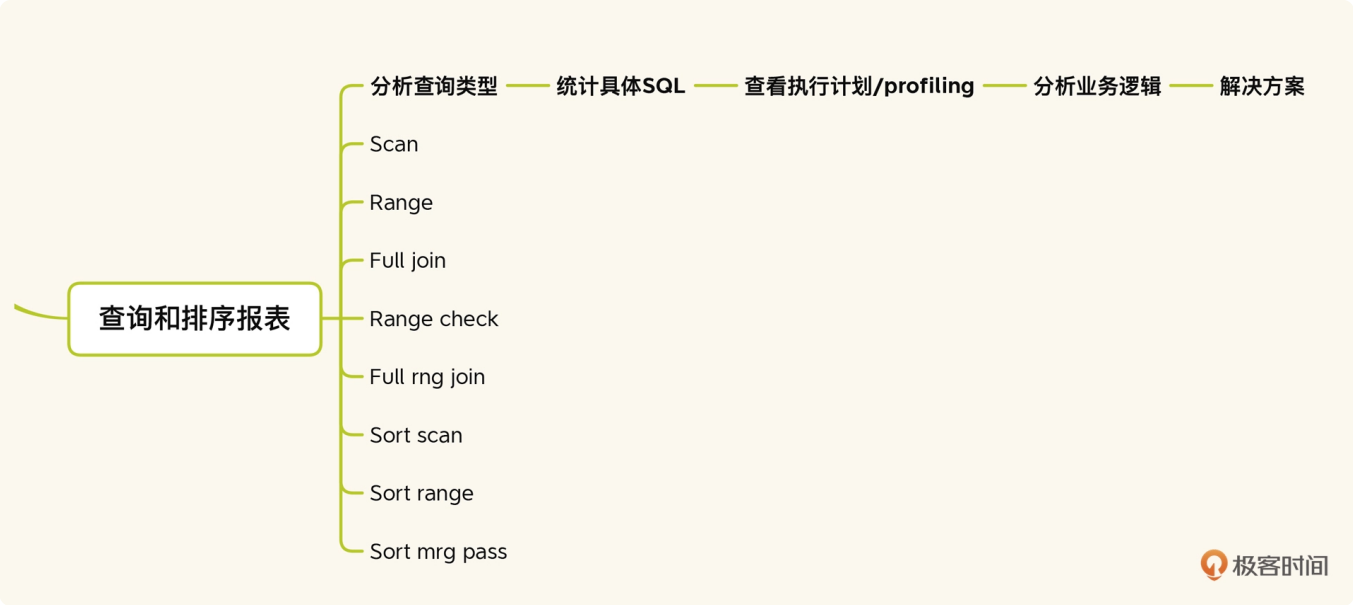
在这里，如果你说：“我以前监控得都不全，加这些怕是不够吧”。那就是你之前的工作有疏漏了，请后退出去把门带上，面壁反省。

分析模型

分析模型是建立在监控模型基础之上的，所以我们必须把性能分析决策树再拿出来，接着往下拆解。

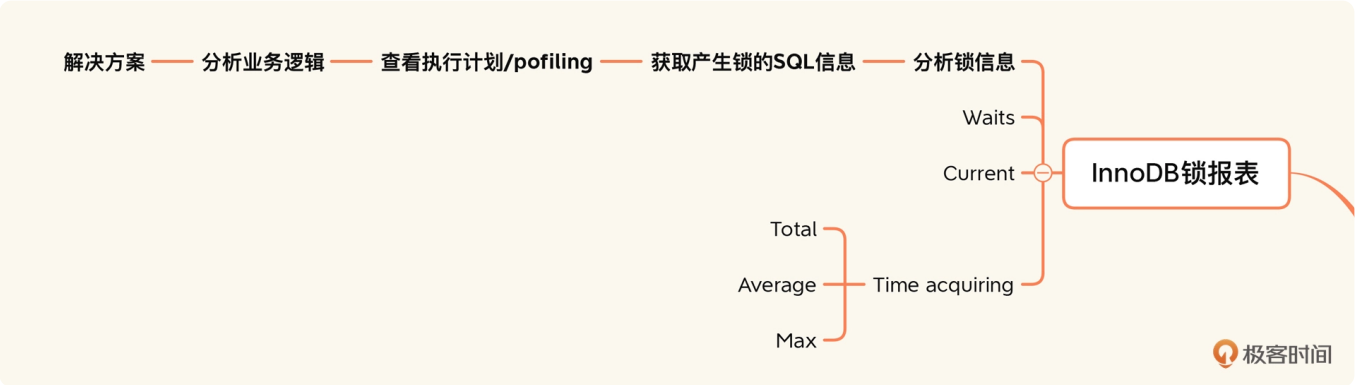
我们来举例说明一下：

对数据库的查询和排序报表类计数器的分析



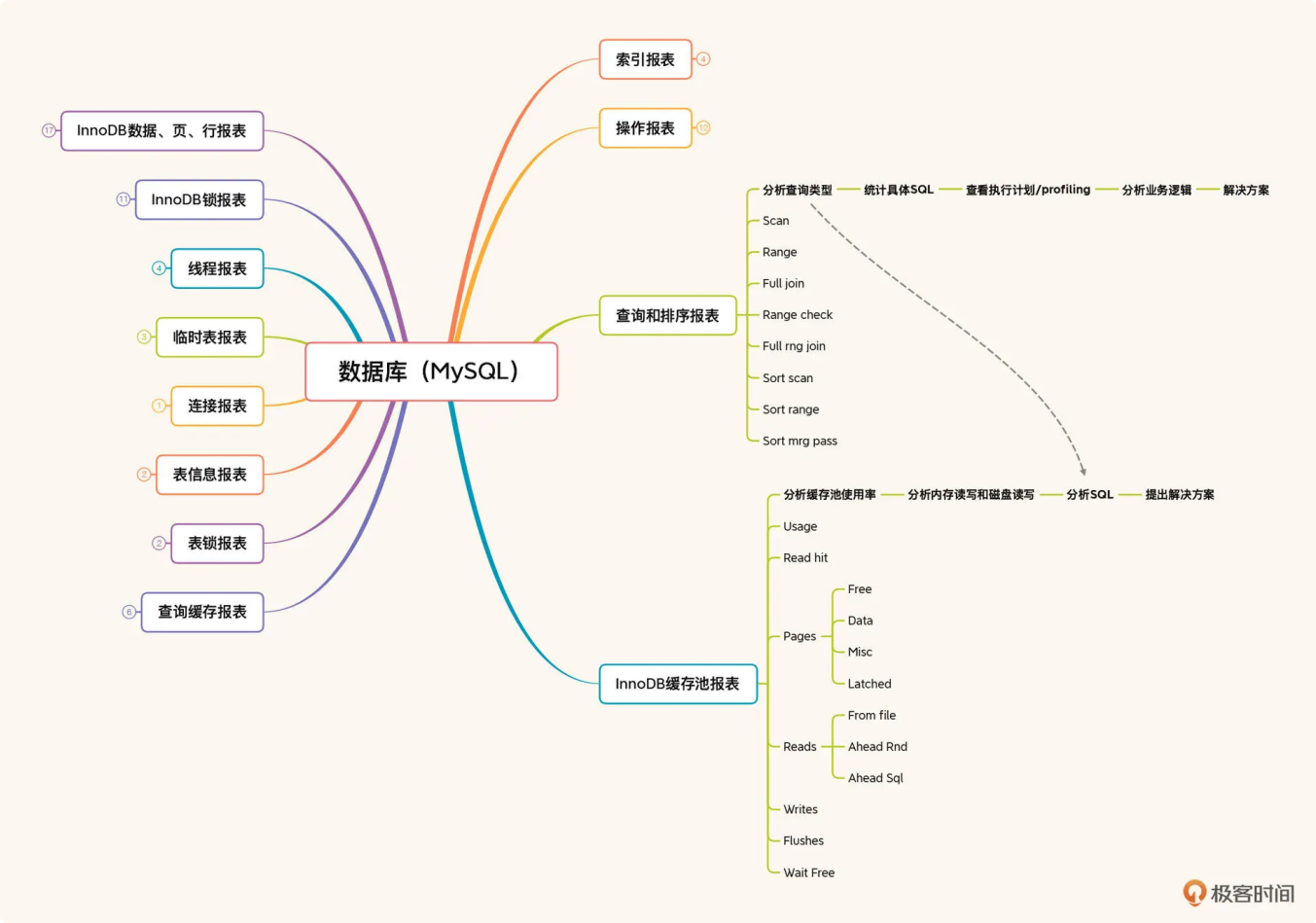
我们要做的是：分析查询类型 - 统计具体的 SQL 语句 - 查看执行计划和 Profiling 信息 - 分析业务逻辑 - 提出解决方案。

对 InnoDB 锁报表计数器的分析



我们要分析的逻辑是：分析锁信息 - 统计具体的 SQL 语句 - 查看执行计划和 Profiling 信息 - 分析业务逻辑 - 提出解决方案。

对 InnoDB 缓存池报表计数器的分析

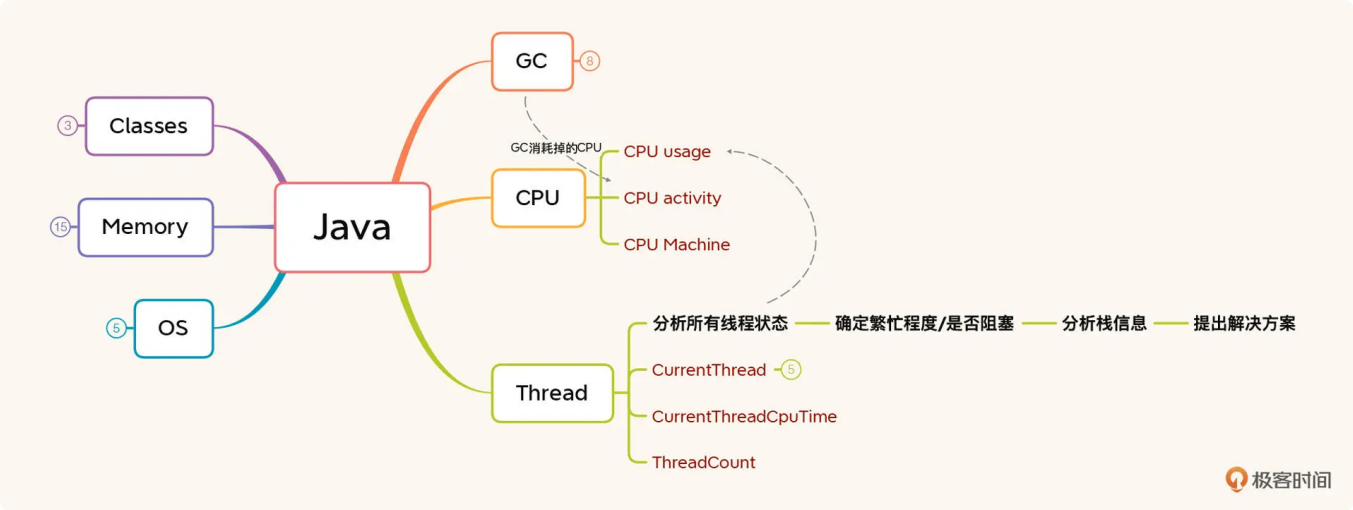


之所以这个图截得大一点，是分析 SQL 和查询排序报表之间有了关联关系。

我们要分析的逻辑是：分析缓存使用率 - 分析内存读写和磁盘读写 - 分析 SQL（分析 SQL 的类型） - 提出解决方案。

再拿 java 的性能分析决策树来举个例子。

对 Java 线程类计数器的分析：



对 Java 的线程池，分析过程是：分析所有线程状态 - 确定繁忙程序 / 是否阻塞 - 分析栈信息 - 提出解决方案。

但是因为分析线程状态要结合 CPU 使用率来分析，所以在图片中你可以看到，他们之间也有一条线。

把所有的分析路径都分析出来，我们的分析模型就创建完成了。其实在这一步，最需要的就是技术功底和耐心。

可能有人会说，这么庞大的分析模型是一个人能完成的吗？根据我的经验，如果你认真学习，多则十年八年，少则五六年，确实是可以一个人完成的。

不过，其实你也不用这么纠结是不是一个人，毕竟性能项目从来都不是一个人的事情，而是一个团队、一个企业的事情。

由于每一类、每一个计数器都需要创建一个分析逻辑，而且往往还需要多个计数器关联分析，所以很难在文章里让你看到我们这一步做得有多壮观。如果我把所有的分析路径都创建出来放在文章里，那图片会糊得像马赛克一样。

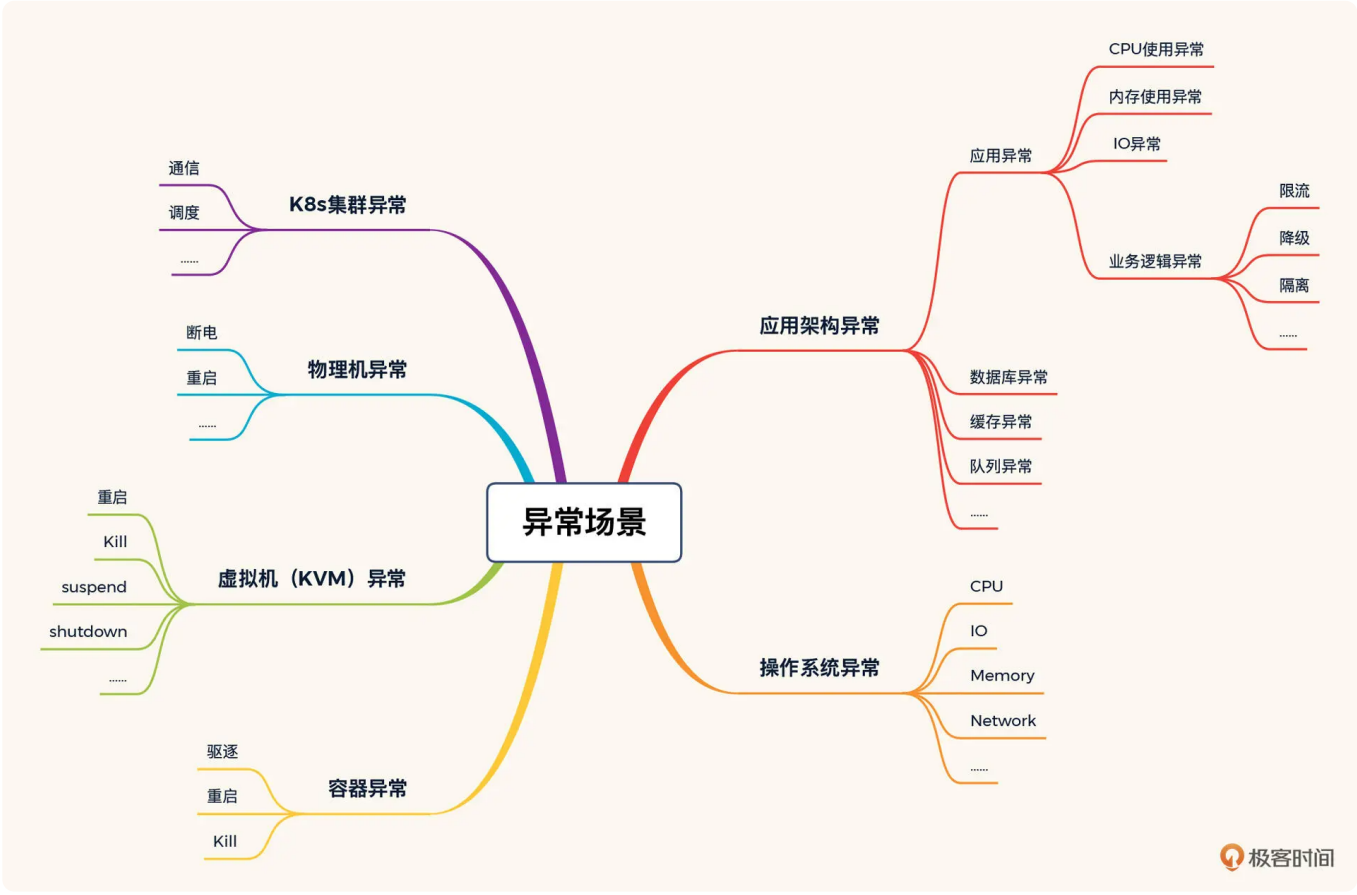
不过，我还是建议你自己去创建针对项目的完整的性能分析决策树和分析模型。因为只有亲手做一遍，才能慢慢理解它的博大精深，也能知道自己会卡在什么地方。

异常（失效）模型

最后，我们再看下异常模型。异常模型又叫失效模型，其实在性能领域中，我真是不想提它，但又不得不提。

在很多企业中，异常、故障、失效这样的场景，都被列为非功能测试范畴，而非功能的案例在性能领域中其实只会蜻蜓点水似的做上几个，不完整不说，效果也很一般。如果想做得完整呢，那就不是一个性能项目的成本能 cover 得住的了。

为什么成本这么高？我们先来看一下异常场景有哪些类型。



在这个图中，我根据技术栈从下到上的层级罗列了一下，大概有 20 多个测试点。请注意，这里的测试点只是针对一个节点来说的，如果你有成百上千的节点，那就是乘法关系。想一想吧，要做完整的异常场景，靠手工是很难做得下去的。

这个时候，你是不是自然而然地想到了“混沌工程”这个概念？

确实，你可以借鉴混沌工程的理念和工具来完成其中一部分内容。但是当前市场上几个用得比较多的开源混沌工具，基本上是停留在微服务分布式应用、容器编排、操作系统这几个角度，而且部分模拟的手段和真实生产环境中出现的故障并不一致。

举例来说，在阿里开源的 ChaosBlade-Operator 中，模拟 CPU 使用率高的方式是启动了一个进程来消耗掉 CPU，其实跟你写一个 do while 或者 sleep 是同样的功能。

复制代码

```
top - 11:13:32 up  1:29,  1 user,  load average: 1.28, 0.58, 0.36
Tasks: 130 total,   2 running, 128 sleeping,   0 stopped,   0 zombie
%Cpu0  : 78.4 us,  2.3 sy,  0.0 ni, 18.9 id,  0.0 wa,  0.0 hi,  0.3 si,  0.0 s
%Cpu1  : 77.9 us,  3.3 sy,  0.0 ni, 18.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 s
KiB Mem :  8008972 total,  2815456 free,  1042748 used,  4150768 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  6657892 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
26633 root      20   0  108552  12488      4 R 147.8  0.2   0:40.34 chaos_burn
 2450 root      20   0  830076  93308  29864 S   1.7  1.2   1:48.33 dockerd
 1459 root      10 -10  193480  25652  14620 S   1.0  0.3   0:57.01 AliYunDun
 3500 root      20   0 1594788  81096  35772 S   1.0  1.0   1:06.78 kubelet
 1949 root      20   0 1172680  52168  14772 S   0.7  0.7   0:06.45 containerd
 4881 root      20   0 1041280  40388  17168 S   0.7  0.5   0:31.43 calico-nod
    9 root      20   0       0      0      0 S   0.3  0.0   0:02.45 rcu_sched
 3770 root      20   0  107688   6492   2784 S   0.3  0.1   0:00.09 containerd
 4811 root      20   0  109096  10564   3100 S   0.3  0.1   0:02.88 containerd
```

你看，上面的 chaos_burncpu 进程就是来干这个坏事的。

有过线上运维经验的人可以回忆一下，在线上遇到过的 CPU 使用率高的性能问题，是不是很少是因为新产生了一个进程而抢掉了 CPU 导致的？通常情况下，CPU 消耗高，都是一个正在使用的应用进程内部产生的性能缺陷导致的，这才是常见的性能缺陷。所以这种模拟手段，并不能覆盖我们在线上看到的问题，只能起个心理安慰的作用。

那应该怎样设计异常模型呢？我的建议是，把你的生产环境中出现的问题罗列出来，分析原因，把相应的缺陷实现到异常模型中去。比如说，由于某个方法写得不好而导致了 CPU 的长期消耗，那你可以写一段方法性能差的代码打包到应用中去，通过开关触发；或者直接 attach 到正在运行的 Java 进程中，将消耗 CPU 的代码直接注入到进程中去并运行起来。这样的异常模型才是符合真实缺陷的设计。

如果针对异常场景的每一个分类，我们都做出符合真实场景中的缺陷的设计，就可以得到完整的异常模型啦了。

总结

你也看到了，这节课，我并没有给你一套拿去就能用的压测模型。当然，这也是做不到的，因为模型只有在一个具体的项目中去细化和创建才有价值。

相比拿来即用的压测模型，我更想传递给你的是思考的逻辑。我希望在这样的逻辑指导下，你可以去创建属于自己的模型，如果有困难，可以来找我沟通。

好了，我们最后总结一下知识点。这节课，我们重点分析了业务模型、容量模型、监控模型、分析模型和异常模型：

业务模型的重点是，根据全链路压测的目标来确定如何实现业务的比例关系。你是要复现历史的容量峰值场景，还是推算未来的容量峰值场景？这个问题很关键，而具体使用什么样的压测工具反而不是重点。

容量模型的重点是，计算的逻辑要合理。你可以单点估算，也可以用混合容量估算。但前提是，一定是符合具体项目的估算逻辑的。我看到有些企业用“TPC-C”或“日交易量 x 峰值系数 / 交易时长”来估算 TPS。在我看来，还是太单薄了。

监控模型的重点是“全”。现在很多企业用云服务器，厂商会提供一些监控视图。在我的经验中，厂商提供的监控视图就没有完整的，通常都是分析时发现缺这少那的。所以别觉得有了监控视图，你就有了完整的监控模型，那是不现实的。还是创建自己的性能分析决策树，再来比对一下缺少什么更为合理。

分析模型的重点是分析逻辑，而分析逻辑靠的是技术功底。所以要想创建分析模式，当能力不足时，可以借助团队的力量。

异常模型的重点是“真实”，不真实的异常，做得再多也没有意义。

在性能领域中，我把以上的内容称为性能五模型，在全链路压测模型的创建中，也是这五个模型，不会有偏差。

课后题

在结束今天的学习之前，我还想请你思考两个问题：

1. 尝试在自己的项目中创建监控模型和分析模型，告诉我你创建时的难点在哪里？
2. 异常模型可不可以独立到压测项目之外进行？

欢迎你在留言区与我交流讨论，我们下节课见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 1　提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇　08 | 基础设施：全链路压测的环境有什么特点？

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。